IT 11 006
Examensarbete 30 hp
Februari 2011

Evaluation of RDBMS for use at Klarna

Chen, Xiaoyu

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Evaluation of RDBMS for use at Klarna

Chen, Xiaoyu

Corporate data plays a very important role in the business processing nowadays. It is not only used as the mediate that records the personal information for the business organization, but also keeps tracks of the valuable data about its customers and transaction history. Any wise company has to make sure that its business data is safely and correctly stored. There are many strategies used to guarantee the data security and correctness. Referential integrity (RI) verifications as a one way is widely adopted in the database system to make sure the data is correct and consistent.
RI is a property of data, which is usually implemented as constraint within a relational database management system (RDBMS). It is used to establish relationships for two referenced tables in RDBMS, such that a column of a table in a declared relationship could only contain the values from the relative column of another table. The goal of RI constraint is to enable RDBMS to have a reliable mechanism which can ensure the database integrity and data consistency when doing a insert, update or delete into a database and none of incorrect manipulation is accepted. Most mature and successful commercial database products such as Oracle and Microsoft SQL Server support the definition of RI constraints internally; On the other hand, the one which does not support to enforce RI rules can be only called a database management system (DBMS) such as Mnesia. Mnesia is an Erlang implemented distributed DBMS and especially designed for telecommunication applications which require fast real-time and non-stop services. However in the domain of telecommunication applications, RI constraints were not critical features for the database system so Mnesia left them to the programmers. In this case, RI constraints have to be implemented outside Mnesia.

Klarna AB is a financial company who supplies easy payment solution for online customers. Kred system is an Erlang implemented product which is running by Klarna AB for dealing with their daily business transactions. In this Kred system, Mnesia is used as the DBMS and the RI constraints are implemented into the application layer of the system. It is a stable system however the RI constraints implementation strategy is not efficient. Hence in the following thesis, a research is investigated to find out a new solution for this problem. In this new solution, the RI constraints implementation will be separated from the database layer. A new layer rdbms with RI enforcement will be established and inserted into the middle of the application layer and the DBMS layer. The reason for calling the new layer rdbms is that a new software called rdbms will be used to implement this new layer. Software rdbms is an Erlang implemented module which could assist Mnesia to do data type and RI constraints verifications. Hence the thesis will introduce the procedure of integrating rdbms into Kred system, programming on rdbms in order to implement RI constraints for Kred system. Later, an evaluation will be carried out to see if software rdbms is mature and efficient enough to be integrated into the live system.

Handledare: Jonas Bergsten
Ämnesgranskare: Kjell Orsborn
Examinator: Anders Jansson
IT 11 006
Sponsor: Erik Stenman, Nicolae Paladi, Uwe Dauernheim, David Bingham

Tryckt av: Reprocentralen ITC

Content

Introduction 7
1.1 Research background 7
1.2 Tasks of the thesis 8
1.3 Outline of Thesis 9
Theoretical Basis 10
2.1 Referential integrity 10
2.2 Data type 11
2.3 Where to implement RI rules 11
2.4 Strategies for implementing referential integrity 12
2.5 Selecting a solution for this thesis 14
Rdbms 15
3.1 What is rdbms? 15
3.2 How rdbms works 16
3.2.1 Workflow 17
3.2.2 Rdbms objects 23
3.3 Why rdbms 24
Implementation environment 26
4.1 Erlang 26
4.2 Kred system 27
4.3 Mnesia 28
Prototype 30
5.1 Design for a prototype 30
5.2 Rdbms API functions 32
5.2.1 set_property(Tab, Key, Value) 32
5.2.2 type(Tab, Attr) 34
5.2.3 tab_references(Tab) 34
5.2.4 drop_references(Tab) 34
System Integration 35

6.1 Design for integration 35
6.2 Rdbms integration details 36
Testing and Evaluation 39
7.1 Correctness testing 39
7.2 System performance evaluation 43
7.3 Mnesia Benchmark evaluation 45
7.3.1 Introduction 45
7.3.2 Benchmark evaluation with rdbms 48
Conclusion 50
Acknowledgement 51
References 51

Chapter 1

Introduction

1.1 Research background

Klarna AB [1] is a financial company created in 2005. It provides an easy payment solution for a number of online customers who open the web stores on the Internet. The numbers of their customers has boomed in recent years. Klarna AB developed its own system Kred which deals with high volumes of small transactions of trading everyday. Erlang/OTP [2] is the system programming environment. Erlang is a functional programming language which was developed by Ericsson [3] and it is designed especially for developing distributed telecommunication projects. An Erlang embedded database Mnesia [4] is used as the DBMS in Kred system. Based on Mnesia, the Kred system can supply non-stop and rapid search reply services for a big group of online customers who make their online purchase day and night.

The reasons of Kred system selecting Erlang/OTP as the programming environment are firstly because Erlang makes the programming easy and save. Erlang is especially designed for providing accurate and non-stop services within the distributed network environment. Hence it is suitable for the case of use of Kred system; furthermore database Mnesia is characterized of providing multi-users response and having fast speed when doing key value look up. Kred system needs these functionalities. Beside this, Mnesia is also powerful to cope with complex objects and it supports hot swapping [5]. However, on the other hand, Mnesia as mentioned above does not support definition of database constraints especially Referential Integrity (RI) [6] constraints which requires the value of a foreign key [7] in a table should match the value from another column of its parent table [8] in the database. As a result, these constraints need to be implemented somewhere else.

In the current Kred system, database constraints such as RI constraints are spread out in different parts of the application objects and processes. It is no doubt that these constraints could guarantee that the database always stays in a correct and consistent state. However, it caused many problems. Firstly, there is a big possibility that constraints defined in completely different objects, processes or written by different programmers may conflict with each other. These conflicts

are hard to be detected. There is also a potential risk that the system rejects valid database manipulate request. Secondly if these constraints are not implemented in one common place, they will become invisible and hard to understand. Code is difficult to adjust since we always have to take many parts and objects into consideration when we want to make a change even it is a tiny one. Unnecessary code redundancy may easily occur in the system.

1.2 Tasks of the thesis

Due to the problems found in the system, some research is needed to find out a new solution. Because the name of the software ‘rdbms’ can easily confuse people with another concept relational database management system (RDBMS). In the rest of this thesis, word with lower case ‘rdbms’ will be only point to the software; RDBMS with upper case of the letters will be the concept relational database management system.

In this new solution, a new software rdbms is used to construct a new layer into the Kred system. According to the description from rdbms user manual [9], rdbms supports two sorts of database constraints definition. One is data type [10] constraints, for instance, a name is string and an age is integer number. The other is RI constraints. rdbms could assist Mnesia to check these data validations.

Hence the thesis will introduce a method of integrating rdbms into Kred system, implement constraints verifications for Mnesia and at last, evaluate the new system in order to see rdbms’s performances. Certain tasks are separated as shown below:

- Understand the mechanism of rdbms
- Understand how to use rdbms API functions to define constraints; understand how to call rdbms API functions to check data constraints
- Design a method to integrate rdbms into the current Kred system
- Integration implementation according to the method
- Design and carry out evaluation on both old and new system and later measure the results
- Conclusion

1.3 Outline of Thesis

In Chapter 2, at the beginning, the thesis lists and explains a set of important concepts involved in the study, including the definition of data type and referential integrity. Later it introduces serveral popular strategies of implementing RI constraints. Finally, a proper solution is selected from these strategies after considering the Kred system's architecture.

Chapter 3 explains the software studied in the thesis, rdbms in detail. At the beginning, there is a brief introduction about rdbms. Afterwards it explains the internal functionality of rdbms including how rdbms stores the data constraints and what happens when rdbms checks the data validation.

Chapter 4 gives a brief introduction about the programming language Erlang/OTP; the framework of the current Kred system and the DBMS adopted by the Kred system, Mnesia.

Chapter 5 introduces a prototype implemented for the thesis before the system integration. From which I practically learned how to use rdbms. In this chapter, the basic design idea about this prototype is explained at the beginning. Then the implementation details are described.

Chapter 6 introduces the integration process about rdbms including the designing for the integration; the implementation of integration according to the design; Code adjusting and changing into the Kred system; Problems solved during the integration.

In chapter 7, a test is carried out to find out whether rdbms really works. Two evaluations are executed to compare the new Kred system performance with the old one.

Chapter 8 makes the conclusion for the thesis and then suggests future work.

Chapter 2

Theoretical Basis

This chapter explains several important concepts relating to the thesis work. Some of them such as strategies for implementing RI rules are the theoretical foundations for the thesis. At the end of this chapter, a proper solution is selected according to the specific demands from the system involved in this thesis.

2.1 Referential integrity

RI is a property of data when satisfied, it requires that the values of the foreign key in one table should target to the values in another column, usually a primary key of the parent table. Figure 2.1 shows an example of referential integrity. There are two tables, one is called employee and the other is called department. Attribute *department* is the foreign key and it targets to attribute *id* which is a primary key belongs to the table department. From the figure, we can see that all the values of attribute *department* are existed in the table department. If in the table employee, there is a record contains a department which is not appeared in table department, for instance, a record *{1004, Bob, Male, 3}*, 3 is not appeared as an element in attribute *id*, that means the referential integrity is violated.

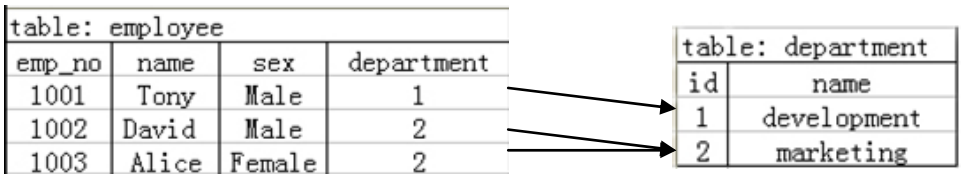


Figure 2.1 Referential Integrity

In a RDBMS system, RI is enforced by respecting the following rules:

When inserting a new record into the table employee, this new record should not have an attribute *department* N when N is not existed in attribute id. Otherwise the insert is not accepted.

When a record is updated in the table department, if the value of attribute *id* is changed, the value of attribute *department* of the relative records should be also changed.

When a record is deleted from the table department, its relative records in the table employee should be also deleted.

2.2 Data type

A data type is another property of data which defines which classification the data belongs to. A data type could be a string, an integer or an atom. Most of programming languages have the notion of data types. For instance, in statically type languages such as Java and C, developers are able to define the data type for variables and later use them within the code. In Erlang/OTP, there is no need to declare the data type of variables before using them. Instead the programming environment supports the ability to dynamically determine the data type for a variable according to the value the developer assigns. Functions such as is_integer(Attr) and is_list(Attr) are capable of judging if Attr belongs to a specific classification or not. Boolean result ‘true’ or ‘false’ is returned back to the tester.

2.3 Where to implement RI rules

Basically there are two choices as to where to implement RI rules. The first and the traditional one is to implement it within database; and the other one is to enforce RI constraints within application logic [11]. However it is not a black and white issue to determine which approach is correct and which is wrong. Both of these two approaches have their own advantages and also drawbacks. None of them are perfect solutions. Before determining an approach for the system implementation, you have to think about the architecture of your building system first and then to analysis if the approach selection really works for you in practice.

RI rules can be implemented within database because modern RDBMS systems such as MS SQL Server and Oracle support complex and powerful functionality for RI definition. RI rules can be centralized controlled and enforced by the database and they are also easy to be adjusted for the future. Hence database is a ideal place to define RI rules. But this approach is only fit for the situation when the application has single database and all the application objects respect to the same RI rules. It is not fit for the modern distributed database environment, for

instance, a situation described in [12], tables with similar contents used by the application may come from completely different database sources, but the referential integrity rules in each database may also not the same. In this case, referential integrity may be easily violated if RI rules are implemented within database.

RI rules can be also implemented within application logic because database development becomes more complex in modern times. RI starts to become a business issue but not only for database so it may be implemented into a business layer. This approach is especially fit for the distributed database environment in which several RI rules are defined within application to satisfy different requirements from the database sources. However because RI rules may be implemented into certain number of objects within application, it will hide the RI rules to be seen. RI rules may be also difficult to be changed because many parts in the application have to be taken into consideration before you commit the changes. Therefore conflicts may easily happen in RI implementation. In this approach, the database can not be used without the application, otherwise it is useless.

In conclusion, neither of these two approaches is perfect. The right choice is what really helps for the system implementation. Sometimes, in a single system, different application components may have different RI implementations.

2.4 Strategies for implementing referential integrity

From the practical point of view, there are several strategies to implement referential integrity. The most popular and traditional strategy is database constraints [13] which is also known as declarative referential integrity (DRI). DRI is to define and enforce RI rules within database by using SQL database programming languages in order to ensure the database integrity and table structures. This strategy is to implement RI rules within database. For example, in MS SQL server, a specific RI rule is established by defining a pair of primary key constraint and its relative foreign key constraint.

Database triggers [14] is another strategy to implement RI rules within database. It is a block of SQL code or other programming language code which is executed when an event occurs into database. The event should be a particular data manipulation transaction such as an insert, a delete or an update. Although database trigger is usually used to implement complex business logic or keep track of database manipulations, RI rules can be also implemented by database

triggers. For example, the syntax for creating a database trigger rule on Starburst relational database system [15] at the IBM Almaden Research Center is:

create rule *rule-name* on *table-name*
when *transition-predicate*
[if *condition*,]
then *action-list*

The *transition-predicate* specifies the event to evoke this database trigger. It could be inserted, deleted or updated. Once a rule is triggered, it will check if the current state of database satisfies the *condition* rule or not. If the condition evaluates to be true, the *action-list* is executed. We can use it to define a particular RI rule on Starburst system. To assume we have two tables, an employee and a department, we want to enforce a RI rule by which if we delete a record from table department, all the employees who work for this department have to be also removed from table employee. Then we define the database trigger as:

```
create rule del-cascade on department
when deleted,
then 'delete from employee
         where employee.department in
         (select d.id from d as (deleted())'
```

There is no condition for this rule, so the action is executed whenever there is a delete into table department. A RI rule can be implemented like this. Compared with database constraints, database trigger is able to define more complex RI constraints.

Persistence framework [16] is a strategy to implement RI rules within application. It is an approach to transform the data from RDBMS system to another format of data and store them into an object which can be widely used by other applications. During this process, RI rules are defined in the meta data as part of relationship mapping. ObjectRelationalBridge (OJB) [17] is a typical Java persistence framework tool which provides its application with transparent access to the relational database. This strategy helps to assemble RI rules into a single meta data repository outside the database instead of spreading them into many parts of business objects. However, it requires that every application must base on the same persistence framework.

RI rules can be also implemented as high level active rules [18] on top of the database. In this strategy, instead of implementing database triggers inside database, RI rules can be implemented by another deductive database language as executable trigger monitor programs within applications. For instance, in [18] the author uses a language called RDL1, implements active rules into a trigger

monitor layer on top of database layer by which RI rules are enforced. The programs can be automatically evoked and executed according to the database manipulation operations performed in the applications. This strategy can be taken advantage of when the database does not support RI definition or RI rules can not be sufficiently expressed inside database.

2.5 Selecting a solution for this thesis

There is no forever optimal strategy of implementing RI rules in a database application. In order to select a proper solution, people have to understand the system architecture first. Sometimes several strategies could be mixed together to create a better solution.

According to the reality about the Kred system involved in this thesis, Mnesia is used as the DBMS. Because it is not supported to define RI constraints inside Mnesia, there is no choice but to implement them within application. Software rdbms relies heavily on Mnesia functions and it is designed in the aim of assisting Mnesia to enforce RI rules. Hence rdbms can be used as a deductive database language for Mnesia by which RI constraints are implemented as a executable program module on top of database. Strategy of implementing high level active rules would be a best choice for Kred system. In Chapter 6, rdbms implementation and integration will be illustrated in details.

Chapter 3

Rdbms

3.1 What is rdbms?

Rdbms is a Erlang/OTP implemented software which was written by Ulf Wiger [19]. It ensures the data validation of Mnesia update including data type and RI constraints verifications and it works as a Mneisa activity [20] callback module. The Mnesia activity is a mechanism when a list of function objects is going to be executed by Mnesia. Functions such as mnesia:lock/2, mnesia:write/3 and mneisa:delete/3 will be forwarded to the Mnesia activity callback module in the context of transaction. The Mnesia activity callback module is shown as AccessMod in the figure 3.1 below. As a result, the Mneisa activity callback module will execute these table manipulation functions as AcessMod:lock, AcessMod:write and AcessMod:delete instead of Mnesia. Figure 3.1 vividly explains how Mnesia activity works. ActivityId represents a unique transaction context for the executing function objects.

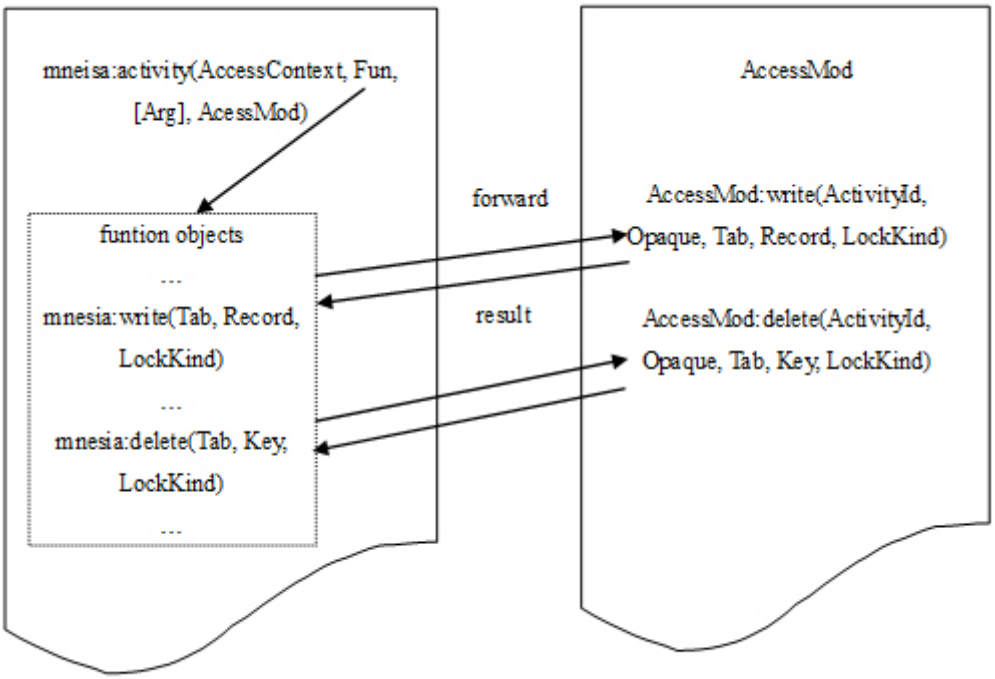


Figure 3.1 Mnesia activity process

The aim of executing functions within a transaction context is to satisfy a set of properties which are called ACID in order to guarantee that the database is updated correctly and reliably. According to the definition of Atomicity-Consistency-Isolation-Durability (ACID) [21], *A* stands for atomicity, it requires that either functions updating the database are executed completely or not at all. Partial database updates are forbidden. *C* means consistency. It requires the transaction to update the database from one consistent state to another. If any one update fails, the entire transaction should be aborted completely. *I* stands for isolation, if a table is updated by a transaction and not finished yet, no other transaction is authorized to access this table until the previous transaction is done. Also all reads should not be able to access a being write object until the write transaction is committed. *D* stands for durability. It requires that the database is able to recover the committed transaction if there is a system crash.

3.2 How rdbms works

In 3.2.1 it explains the internal workflow of different rdbms events. These events include a definition for a specific data constraint and a look up for a data

constraint. In 3.2.2 a set of rdbms objects are listed with the explanation about how these objects are used by rdbms.

3.2.1 Workflow

The workflow of different events of rdbms are going to be explained in this section, including what happens when define a data type into rdbms, define a RI constraint into rdbms, look up a data constraint from rdbms, insert a new object into the table and delete a object from the table.

- Define a data type

A new data type can be defined by calling rdbms API function `set_property(Tab, {attr, Attr, type}, Type)`. Three important parameters should be past to rdbms which are `Tab`, a tuple `{attr, Attr, type}` and `Type`. `Tab` is a table name which tells where the executing attribute comes from; the tuple `{attr, Attr, type}` tells that the achieving goal is to define a data type for attribute Attr; `Type` tells the data type to be defined. This function is to define the data type of attribute Attr. In figure 3.2, it shows the workflow in which three steps are carried out for this rdbms event.

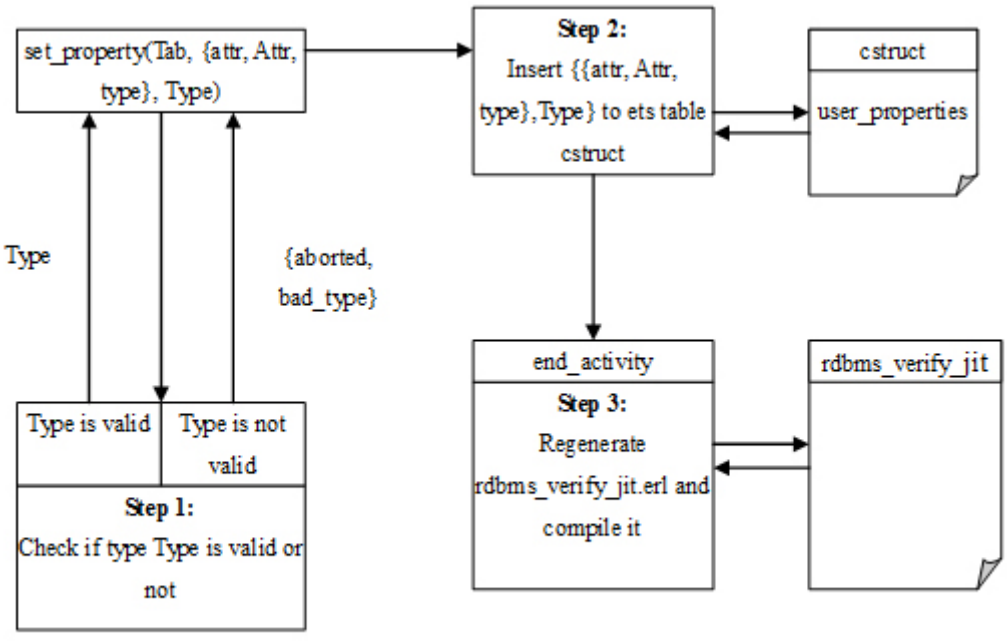


Figure 3.2 Workflow of defining a data type

There are three main steps when defining a new data type. In the first step, `Type` is examined to make sure that it is recognizable for rdbms. The value of `Type` must belong to a value list which is internally prescribed in rdbms. This list will be described later in 3.2.2. In the second step, rdbms inserts a new tuple into an ets table which is created by Mnesia at the very beginning when Mnesia is initialized. The format of this tuple is `{{attr, Attr, type}, Type}` in which the new data type is stored. An attribute of this ets table called cstruct which is used to keep the database schema information for Mnesia and the tuple is inserted into a field called user_properties which belongs to cstruct. In the third step, a new Erlang file rdbms_verify_jit.erl is automatically generated by rdbms according to the definition of the new data type. A timestamp is also generated and added to this new file. Then this rdbms_verify_jit.erl is compiled and replaces the old ones.

- Define a RI rule

The event of defining a new RI constraint is similar to the event of defining a new data type. It can be achieved by calling rdbms API function `set_property(Tab, add_references, Value)` in which three parameters are required. The first one is `Tab` which tells the name of the executing table; the second one is always `add_references` which tells the achieving goal is to define new RI constraints into rdbms; the third one is `Value` which is a list explaining the RI constraints. Figure 3.3 describes the workflow for this event.

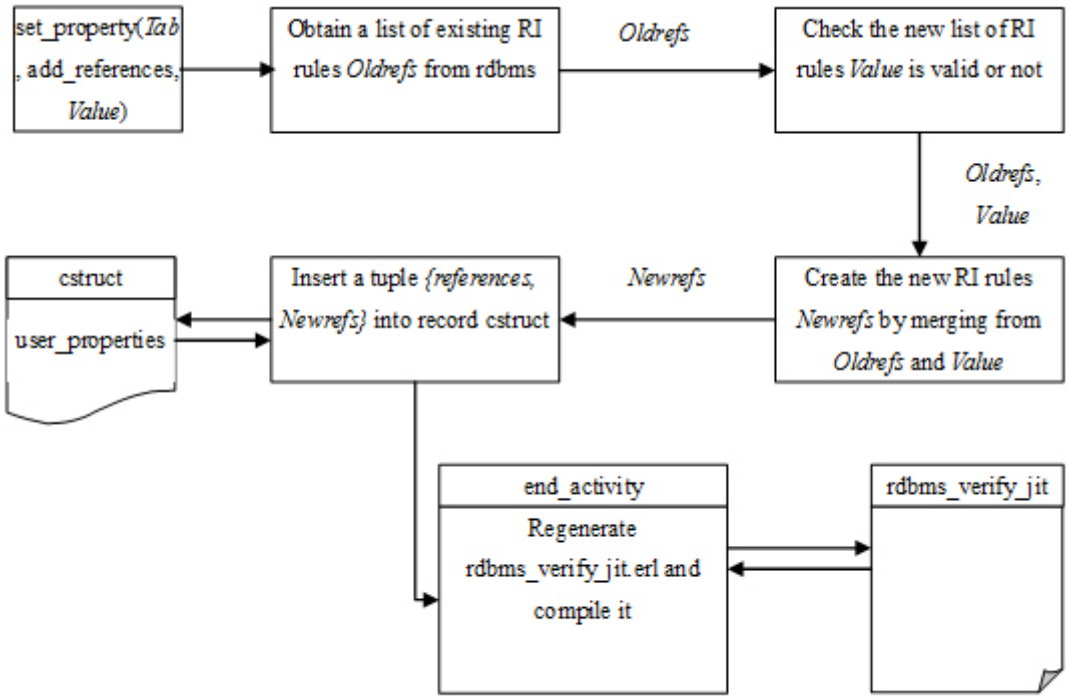

set_property(Tab, add_references, Value)
Obtain a list of existing RI rules Oldrefs from rdbms
Oldrefs
Check the new list of RI rules Value is valid or not
Oldrefs, Value
cstruct
user_properties
Insert a tuple {references, Newrefs} into record cstruct
Newrefs
Create the new RI rules Newrefs by merging from Oldrefs and Value
end_activity
Regenerate rdbms_verify_jit.erl and compile it
rdbms_verify_jit

Figure 3.3 Workflow of defining RI rules

In the first step, rdbms reads the existing RI constraints from cstruct for table `Tab`. Then in the second step, rdbms examines the validation of `Value`. If `Value` passes the examination, in the third step a new list of RI constraints are merged by combining the existing rules and the `Value`. If `Value` fails the examination, the request is rejected. In this step, rdbms is not able to redefine a RI constraint which has existed in rdbms. That means RI constraints can not be updated directly. However we can drop it first and re-define it with the new constraint in order to achieve updating. When it fails, the transaction will be aborted with the reason "ref_integrity conflicts". In the next step, the new generated list of RI constraints is inserted into user_properties of cstruct for table `Tab` in the format of a tuple `{references, Newrefs}`. In the last step, rdbms generates a new file, rdbms_verify_jit.erl according to the changing. This file is automatically compiled and replaces the old ones.

- Look up an existing data constraint

When looking up an existing data constraint from rdbms, the system will fetch the matched constraint based on the request from the field user_properties of cstruct. The returning result is in the format of a tuple.

- Insert a new object into the table

After the data constraints are properly defined into rdbms, we can start to see how rdbms examines database updating according to these constraints. Figure 3.4 illustrates the workflow when an event of inserting a new object into the table happens.

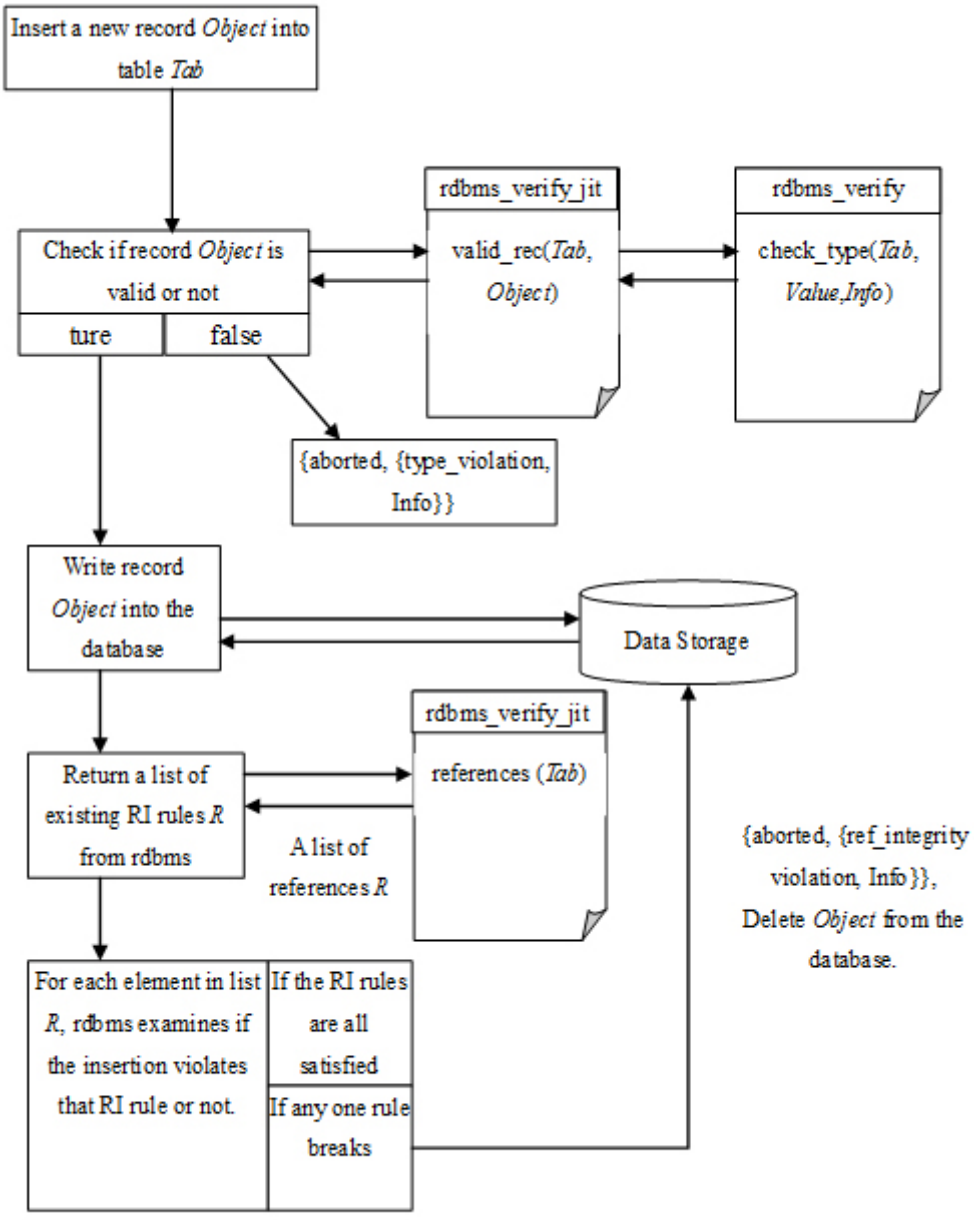

Insert a new record *Object* into table *Tab*
rdbms_verify_jit
valid_rec(*Tab*, *Object*)
rdbms_verify
check_type(*Tab*, *Value*,*Info*)
Check if record *Object* is valid or not
ture
false
{aborted, {type_violation, Info}}
Write record *Object* into the database
Data Storage
rdbms_verify_jit
references (*Tab*)
Return a list of existing RI rules *R* from rdbms
A list of references *R*
{aborted, {ref_integrity violation, Info}}, Delete *Object* from the database.
For each element in list *R*, rdbms examines if the insertion violates that RI rule or not.
If the RI rules are all satisfied
If any one rule breaks

Figure 3.4 Workflow of insert a new record

When a new object is to be inserted into the table, in the first step, rdbms examines the validation of record `Object`. The examination includes checking if the format of the `Object` is correct or not, if the value of every attributes in the Object is correct or not. Two files rdbms_verify_jit.erl and rdbms_verify.erl are involved in the data type violation checks. The former file provides the data type definition for each attribute of the `Object`, the latter file decides if the examination is passed or not. If the examination is failed, the insertion is aborted with the reason "Data type violation"; otherwise a true is returned and it continues for the next step. In the second step, the `Object` is inserted into the table. Then in the third step, rdbms examines if `Object` respects RI constraint of table `Tab` or not. In the beginning of this step, rdbms returns a list of existing RI constraints of table `Tab` by reading from file rdbms_verify_jit.er. For each RI constraint in this list, rdbms matches a list of referenced objects from the table. Then rdbms examines if there is a violation or not by comparing the list of RI constraints and the referenced objects. For instance, a table person has several attributes such as name, age, sex and department. A table department has attributes such as name and floor. There is a RI relation between object person and department in which person's department is the foreign key of department's name. The department's name is a parent key. To assume a new object {person, {"Tony", "25", male, "MIC"}} is to be inserted into table person while there is an object {department, {"MIC", 5}} which has been existed in table department. Rdbms will return a list of referenced objects [{department, {"MIC", 5}}] which means the RI constraint is satisfied. However if rdbms return an empty list from table department, rdbms must detect that this is a RI violation and therefore the transaction must be aborted by rdbms. If it is aborted, an error "ref_integrity violation" is returned and `Object` will be deleted from the table.

- Delete an object from the table

When delete an object from the table, there are several steps rdbms will carry out in order to satisfy the RI constraints. Figure 3.5 shows the rdbms workflow to delete an object from the table.

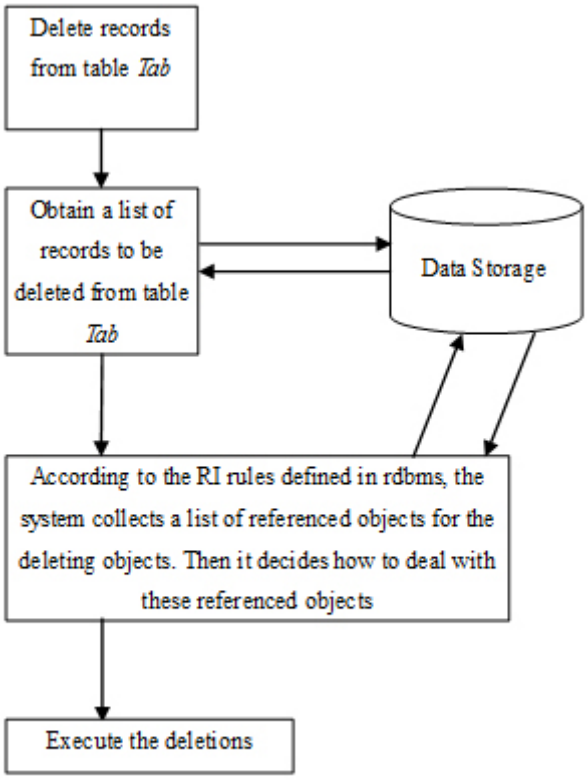

Delete records from table Tab
Obtain a list of records to be deleted from table Tab
Data Storage
According to the RI rules defined in rdbms, the system collects a list of referenced objects for the deleting objects. Then it decides how to deal with these referenced objects
Execute the deletions

Figure 3.5 Workflow of deleting a record

Unlike inserting a new object, deleting an object from the table is sample. In the first step, rdbms returns a list of deleting objects from the table according to the key subscribed in the request. In the second step, a list of RI constraints for table `Tab` is obtained by reading from the file rdbms_verify_jit.erl. For each RI constraint in this list, rdbms matches a list of referenced objects from the tables. In the next step, rdbms carries out a series of executions to the referenced objects according to the RI constraints. Using the same example from the former paragraph, we assume to delete an object {department, {“MIC”, 5}} from table department. Rdbms will return a list of referenced objects [{person, {“Tony”, “25”, male, “MIC”}}] from table person. Because the relationship between table person and department, if the object {department, {“MIC”, 5}} is deleted from the table, the objects related with this department must be also deleted at all. Hence in this step, rdbms will firstly call Mnesia to delete the object of department from the table and then the object of person is also deleted before the transaction is finished.

3.2.2 Rdbms objects

- Cstruct object

Cstruct is the Mnesia's metadata object in which table schema information are stored. These schema information includes fields such as the table name, access mode and database storage. It is automatically generated in the beginning of Mnesia initialization. Rdbms uses one field user_properties to store its additional metadata such as RI constraints and data type as Mnesia. The value of data type is stored in the format of `{{attr, Attr, type}, Value}` and RI constraints are in the format of `{references, [Refs]}` in user_properties.

- Rdbms_verify_jit.erl and rdbms_verify.erl

Both of these two files manage the data constraints in rdbms. rdbms_verify_jit.erl is repeatedly generated to provide the latest data constraints for rdbms. This file is generated by merging the old constraints with the updated ones whenever there is an rdbms event happening which makes a change about the current data constraints. A time stamp is also created based on the right now time and it is added as part of the file name. The new file is to be compiled and replaces the old ones. File rdbms_verify_jit.erl can store the data constraints for tables as well as user_properties does. But they are used in completely different situations. The former one is used as a dictionary for looking up a data constraint according to the request; rdbms_verify_jit.erl is an Erlang module which is used to directly carry out the verifications. It is much faster.

There are also difference between file rdbms_verify_jit.erl and rdbms_verify.erl. We can say that rdbms_verify_jit.erl defines the scenario of verification, on the other hand, rdbms_verify.erl judges if the request passes verification or not. To explain it in another way, rdbms_verify_jit.erl subscribes what to check. For instance, it subscribes to check the value of a personal number P is integer or not. Then it calls rdbms_verify.erl in which it returns true if the value is an integer or false if P fails the examination.

- A list of valid data type

Rdbms internally subscribes a list of identifiable data types. The list is able to be obtained by calling rdbs API function `rdbms_props:simple_builtin_types()`, then a list [any, undefined, atom, integer, float, number, string, text, list, nil, tuple, pid, port, reference, binary, oid, function] is returned. rdbms supports to add customized data types.

- Rdbms:perform_ref_actions/8

This function subscribes the actions how rdbms have to deal with the referenced objects if there is a database update in order to satisfy the RI constraints. Actions can be opted from a list [cascade, set_null, set_default, no_action, ignore]. These actions illustrate how rdbms have to copy with the referenced objects for database updating. For instance, cascade illustrates that if an object is to be deleted from the parent table, a cascade deletion have to be carried out into the foreign table. Action list will be discussed in details in the prototype.

- Compile rdbms_verify_jit.erl

Rdbms_verify_jit.erl is complied by rdbms By calling API function `compile:forms(Forms, [Options])`, `Forms` can be either an Erlang module's name or a piece of abstract format code. `Options` can be report_errors, report_warnings or both of them. Then the result could be either `{error, Information}` or `{ok, rdbms, Binary[,Warnings]}` according to the truth if the code is correct or not. An object code `Binary` is generated if the compiling is done and replaces the old one. No object file is generated. The replace will happen without stopping the running program. Because rdbms_verify_jit.erl must be generated and compiled frequently, it badly affects the performance of the system. Fortunately data constraints are commonly defined once for all at the very beginning of initializing Mnesia.

3.3 Why rdbms

Due to the reason that Mnesia does not support to define the data constraints internally, we have to find some other way to implement them in the application layer. Rdbms was especially designed as a separate call back module which offers definition of data type and RI constraints and it helps Mnesia to execute data constraints verifications. Hence rdbms is a good solution for this problem.

Furthermore, Klarna system Kred is a typical business application in which all data is stored and controlled by a centralized database server node. Besides, all the application objects and processes respect to the same data constraints. Hence to implement RI constraints within a common place in the application layer is suitable for Kred system. In this case, the database application will work efficiently. Rdbms for this case is a wise choice for Kred system. It could assemble both RI constraints and data type definition within an independent layer in Kred system by replacing the current implementation that all of them are spreading out into application layers.

Thirdly, the costing time on integrating rdbms into Kred system could be fast. The reason is that rdbms was especially designed as a Mnesia activity call back

module. The activate rdbms for Mnesia could be easily achieved by writing several lines of code if we use a proper version of Mnesia. Rdbms has no bad effects on Mnesia functionalities and the data is still wholly controlled by Mnesia. Hence developers could save both time and effort required from writing integration codes. Furthermore, there is no need to worry about stability of rdbms integration because rdbms inherits a large number of error handling and exception monitoring mechanisms from Erlang. It assists developers to build a robust system. Additionally, rdbms only generates a small file rdbms_verify_jit.erl to carry out data constraints verifications. It neither creates a new generic server nor runs on a new process in the system. Hence it consumes little time and memory but could respond fast.

Finally rdbms is easy to use because it provides a set of standard API functions for the developers.

On the other hand, rdbms integration requires a proper version of Mnesia; otherwise Mnesia is not able to accept the integration and we need to adjust rdbms. Although rdbms has no bad effects on Mnesia functionalities, it writes Mnesia patches which add new features for Mnesia. When Mnesia is started, these patches need to be loaded as into the system and replace the old ones. In rdbms, all the Mnesia patches modules are included in the folder called mnesia_patches. Hence in order to integrate rdbms into a specific version of Mnesia, a group of Mnesia patches have to be implemented. In this thesis example, Mnesia 4.4.7 was used because the latest package rdbms was implemented based on it. Then there was little change need to be carried out into the original patches.

Chapter 4

Implementation environment

4.1 Erlang

The name Erlang was either given to show respect to a Danish mathematician and engineer Agner Krarup Erlang or can be short for “Ericsson Language”. It is a functional programming language especially designed for implementing telecom applications. The history of Erlang can be traced back to the 1980s, when a group of people including the main developer Joe Armstrong and some other engineers in Ericsson conducted a research in which a series of programming languages are compared to find out a most suitable language for telephone applications implementation. Unfortunately they did not find the best solution and in the end, they came upon the idea of writing their own language, Erlang. Due to the highly concurrent nature of telecom applications, Erlang was implemented to satisfy the requirements for instances, highly distributed and concurrent network environment; large amounts of simultaneous transaction handling ability and the system should never crash; low data losing possibility even a database sever was temporarily not available or the system was being updated. Joe Armstrong wrote the first version of Erlang at the Ericsson Computer Science Laboratory in 1986. Since then a number of useful features have been added to the source code including the important API library Open Telecoms Platform (OTP) which was released in 1996. OTP is a large collection of libraries which supported to solve network and telecommunication problems. As a result, the developers were free to concentrate on solving business logic problems for the application layers. Besides, OTP offered a set of error handling mechanisms, and helps to build a robust system. In the 1998, Erlang became open source for the public. Since that time, the community of user has dramatically increased. Many big companies have selected Erlang/OTP to be the programming environment for their system.

Erlang was characterized by a group of features according to the Open-source Erlang White Paper [22]. The most obvious one was concurrency. Erlang can generate completely distributed processes. Each of them could have its own executed tasks and memory. The memory never leaked to the other processes. None of the processes hindered others’ running. However different processes could communicate by sending and receiving messages.

Erlang could also support hot code swapping which meant the code could be changed and compiled while the system was running. It was a very important feature especially for the real time telecoms applications.

Furthermore, Erlang helped developers to build a robust system since it supported powerful error handling mechanisms.

Finally, Erlang also eliminated the difference between programming environments. Erlang program is capable to be directly executed on a remote node despite the platform or operating system differences.

4.2 Kred system

The Kred system was implemented based on Erlang/OTP in Klarna. Figure 4.1 shows the framework of Kred system. There are three big layers in Kred system.

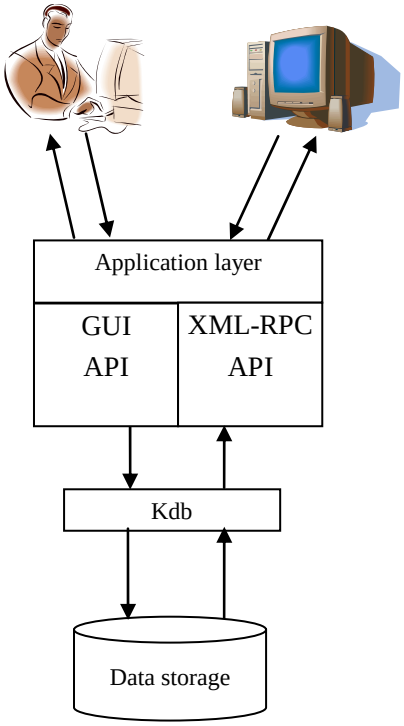


Figure 4.1 Framework of Kred system

The bottom layer is called data storage layer. In this layer, Mnesia is used as DBMS to manage data for Kred system. Above the data storage layer, there is a kdb layer. Kdb layer is considered to be the middleware connecting application

layer and data storage layer. It receives the requests from application layer and forwards them to the data storage layer. Then the database will be updated according to the requests. After the database is updated, Mnesia will send the result back to the kdb layer.

The application layer is the top layer of Kred system. It provides services to the clients both for human clients who subscribe the requests on the GUI webpage and machine clients who connect to the Kred server. In application layer, business logic is implemented in order to check the validation of the coming requests. For instance, the application layer will check if a buyer has good credit history before it allows him to make a new purchase on the web-store. The application layer, on the other side, communicates with kdb layer. If the requests pass the application layer's examination, they will be forward to the kdb layer.

Kred system provides GUI webpage for human clients. People can visit Kred web page to register an account, pay their bill and modify their personal information.

A remote machine needs to subscribe their demands through XML-RPC call. XML-RPC [23] is the abbreviation of XML-remote procedure. It is a protocol to define the method used by a remote machine calling a single method onto another machine. Message of both sending and receiving are encoded according to XML structure. HTTP is used as the transport protocol.

4.3 Mnesia

Mnesia is a multiuser and distributed DBMS which is especially designed for implementing telecom applications. Because it provides fast real time and non-stop services to its users, Mnesia was widely used by a group of Erlang projects such as Kred system which is mentioned in this thesis.

Since Mnesia was originally designed for implementing telecom applications, it introduced a set of concepts of telecommunication into DBMS. Mnesia is also characterized by its powerful functionalities and fault tolerance ability. The powerful functionalities are shown in many aspects. Firstly Mnesia is able to handle many complex values such as lists, tuples, records and even trees thus providing convenience for the developers. This feature is inherited from its implementing language Erlang/OTP. Secondly a developer could easily build a wholly distributed DBMS environment with Mnesia where data can be easily reached and updated remotely. Without being classified into different levels, distributed database servers are equal to each other. Last but not at least, Mnesia supports hot swapping which means that the code can be updated without stopping the system.

Mnesia has also inherited a set of error handling mechanisms from Erlang/OTP which ensures the stability for the application. The wholly distributed database environment also makes Mnesia stronger. If there is a disaster in any one of the database node and it becomes not available for a while, the coming transactions are still acceptable and updated into other nodes. After a while, the crashed node will recover its data from its neighbors.

Chapter 5

Prototype

This chapter introduces a small prototype implementation by which I learned how to use rdbms API functions from the technique point of view. Mnesia was used as the DBMS in this prototype. Rdbms was also integrated into this prototype and it wass responsible for the data type and RI constraints verifications. In 5.1, it explains the designation for this prototype; in 5.2, my learning result from this prototype is reported. In this section, the rdbms API functions are described in details.

5.1 Design for a prototype

The designation for this prototype was firstly to establish a DBMS based on Mnesia in which several entities (tables) and relationships were defined. Then in the second step, rdbms was activated as the Mnesia activity call back module for the DBMS. In the third step, a list of data type and RI constraints were defined for the tables. After the system was started properly, clients were generated for testing the prototype system. For the first time, valid requests were sent from clients to the DBMS. These valid requests were expected to pass rdbms. Then the invalid requests were generated and sent to DBMS for the second time. Rdbms was expected to stop these requests and error message would be returned back to the clients.

A small database was involved into this prototype implementation. Figure 5.1 shows the ER diagram.

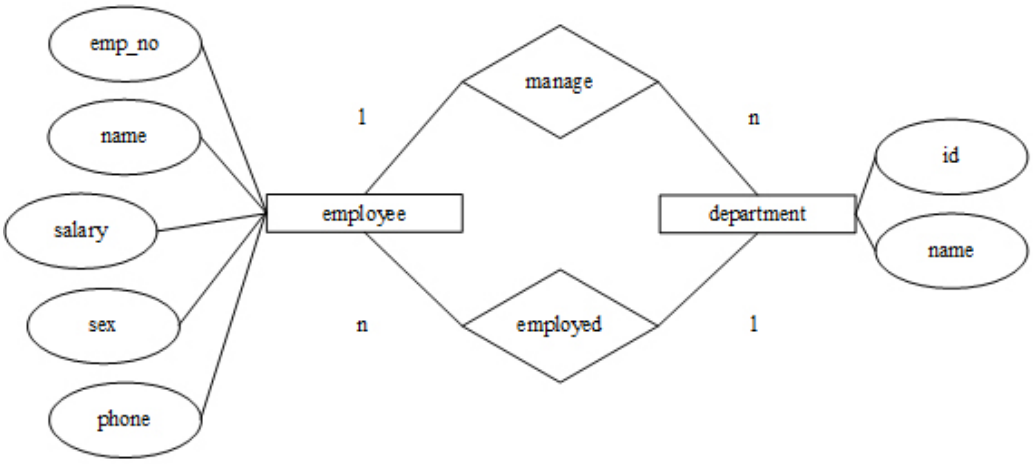


Figure 5.1 ER digram

There are two main entities in the ER diagram, department and employee. There are also two relationships between these two entities. One is “manage 1:n relationship”. It illustrates a relation that on the one side, an employee could be one or several departments’ manager and on the other side, a department has only one manager; The other is “employed n:1 relationship” and it illustrates a relation that on the one side, an employee can be only employed by one department and on the other side, a department can employ one or several employees. Hence four tables are created in the database: employee, department, manager and employed.

Integrating rdbms into the prototype system could be achieved by calling `mnesia:start([{access_module, rdbms}])` in a proper place where Mnesia is initialized.

After Mnesia DBMS was started, the four tables need to be created. Commonly a table was created by calling Mnesia API function `mnesia:create_table(TabName, TabDef)`. `TabDef` was a list of tuple `{Item, Value}` in which the table schema information was stored. Rdbms enabled the developers to define a new tuple `{rdbms, Properties}` for the table. Attribute `Properties` was a list of expressions which defines the rdbms constraints. For instance, the `Properties` could be a tuple `{{attr, emp_no, type}, integer}` which defines a data type constraint for the table. However it was better to define all the constraints after creating all the tables because RI constraints could not be accepted if the relating table was not existed in the database.

Rdbms constraints were defined by calling rdbms API functions. In this part, I confronted with some difficulties to write the correct codes. Firstly, the instruction about rdbms was not good. There was a brief manual, but still not enough. Some of the contents were even out of date. For instance, I found in the

source code, some rdbms API functions were moved to a different module and some of them were removed completely. But the manual was not updated with these changes. Secondly the source code was not well documented. The description beside each function was unclear. I could hardly understand its functionality. Neither could I understand how to set for the input parameters unless I read the source code in details. Unfortunately, it would have great impacts on executing these functions correctly. Hence I spent a certain number of time on reading rdbms code. I also found out the proper type for the input parameters by trying out different ones.

Because of my hard learning, I have known how to use rdbms API functions and successfully defined the rdbms constraints for the four tables involved in the prototype.

Then I tested the prototype system and the testing results proved that rdbms was effective as the expectation.

5.2 Rdbms API functions

Most of rdbms standard API functions are written in module rdbms_porps.erl.

5.2.1 set_property(Tab, Key, Value)

This function is used to add or update data constraints into rdbms. It updates the value of *Key* for the table *Tab* to be *Value*. The parameter *Key* can be any value listed below in order to achieve different functionalities:

{attr, Attr, type}

Then the function *set_property(Tab, {attr, Attr, type}, Value)* is going to define a new data type constraint. Attribute *Attr* is the working on attribute; attribute *Tab* is the name of table where attribute *Attr* belongs to; attribute *Value* describes the new constraint. Attribute *Value* can be any from a list [any, undefined, atom, integer, float, number, string, text, list, nil, tuple, pid, port, reference, binary, oid, function]. For each time, only one data type constraint is allowed to be defined into rdbms.

add_references

Then the function *set_property(Tab, add_references, Value)* is going to add a list of new RI constraints into rdbms. Attribute *Value* is a list of tuple *{attr, Attr, [{Tab2, Attr2, Actions}]}* which describes the new RI constraints. A new

relation is established between the table *Tab* and the table *Tab2* and two attributes *Attr* and *Attr2* are involved. Attribute *Actions* describes this relation in a tuple *{Match, DeleteAction, UpdateAction}* which is a description about rdbms execution in order to satisfy the RI. Three parameters *Match*, *DeleteAction* and *UpdateAction* illustrate rdbms execution which is explained below:

The value of parameter *Match* could be either partial or full which defines how attribute Attr and Attr2 match. It only matters when the matching attributes are compound attributes. A compound attributes maybe a record or other form of complex attribute which contains more than one element. For instance, a person's name is a compound attributes in which three elements are contained: first name, middle name and last name. In the context of compound attributes, if partial is selected, match happens in two cases: 1. Non-null components of *Attr* equals to its counterpart of *Attr2*. 2. All components of *Attr* are null. If full is selected, match happens in two cases: 1. *Attr* is completely equal to the *Attr2*. 2. All components of *Attr* are null.

The parameter DeleteAction describes that if there is an object to be deleted from table *Tab*, the rdbms execution on the related objects in table *Tab2* where *Tab2.Attr2 == Tab.Attr*. Options of DeleteAction are set_null, set_default, cascade and ignore. If set_null is selected, after the object is deleted from table *Tab*, the value of attribute *Tab2.Attr2* in the related objects is set to be null. If set_default is selected, the value of attribute *Tab2.Attr2* in the related objects is set to be the default value. If cascade is selected, the related objects will be also deleted from table *Tab2*. If ignore is selected, rdbms will execute nothing.

The parameter UpdateAction describes that if an object is to be inserted into table *Tab*, the rdbms execution on the related objects in table *Tab2* where *Tab2.Attr2 == Tab.Attr*. Options of UpdateAction are set_null, set_default, no_action and ignore. If set_null is selected, before the object is inserted into table *Tab*, the value of attribute *Tab2.Attr2* in the related objects in table *Tab2* is set to be null. If set_default is selected, the value of attribute *Tab2.Attr2* in the related objects is set to be the default value. If no_action is selected, rdbms will execute a checking first aimed to find out at least one related object in table *Tab2*, otherwise rdbms will reject the inserting request. If ignore is selected, rdbms will execute nothing.

drop_references

Then the function *set_property(Tab, drop_references, Value)* is to delete the existing RI constraints from rdbms. *Tab* describes the working on table; *Value* describes the dropping constraints. If there is no such constraints existed within rdbms, rdbms will execute nothing.

5.2.2 type(Tab, Attr)

This function is to look up a specific data type constraint from rdbms. Parameter `Tab` describes the working on table and parameter `Attr` subscribes the requesting attribute.

5.2.3 tab_references(Tab)

This function is to return all the RI constraints of a table. Parameter `Tab` subscribes the rquesting table.

5.2.4 drop_references(Tab)

This function is to remove all the RI constraints of a table. Parameter `Tab` subscribes the deleting table.

Chapter 6

System Integration

After learning the rdbms API functions, in this chapter, the procedure of implementation of integrating rdbms into Kred system is illustrated. In the first section, the system framework adjustment for preparing the integration in which a new layer rdbms is added into the framework is introduced. In the second section, the implementation details of system integration are described.

6.1 Design for integration

In order to integrate rdbms into Kred system, the system framework must be adjusted for the rdbms integration. The new framework of the system is shown in Figure 6.1.

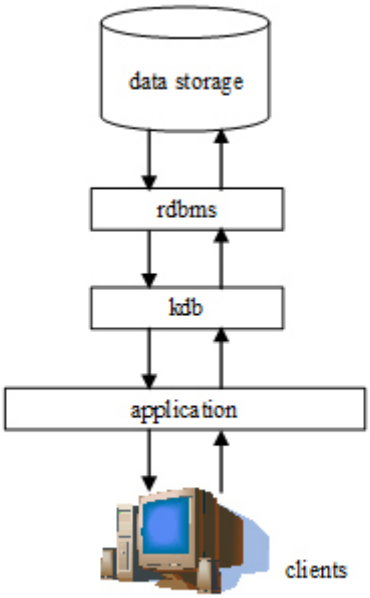


Figure 6.1 New framework of Kred system

In the new Kred system framework, a new layer rdbms is added between the kdb layer and the data storage layer. Rdbms layer is responsible for examining the validation of the requests coming from the kdb layer. The examinations include the data type verification and RI constraints verification. If these requests pass the examination, rdbms will forward them to the data storage layer where the database is to be updated. In the new system framework, the rdbms layer will cut off the connection between the kdb layer and the data storage layer completely. Instead, a new connection is established between the kdb layer and rdbms layer. The data storage layer is no longer handles the requests from kdb layer but the connection makes it to directly communicate with rdbms layer and accepts the requests from it.

6.2 Rdbms integration details

The old Kred system is a robust system but not efficient. Although it is able to detect data type and RI constraints violations, it spreads out the exam code into too many processes and objects within the application layer. Instead of only concentrating on solving business logic problems, the application layer spends time and effort on dealing with database constraints. Furthermore, this will also cause a large amount of useless code redundancy in the application layer.

In the beginning, my original plan for this implementation was to remove the redundancy code from application layer before implementing an extra rdbms into the kred system. The benefit was that it lightened the workload for the application layer, effectively reducing the code redundancy and releasing the application layer to concentrate on business logic problems. Unfortunately, this task was not easy because the code was hidden too deep into many parts within the application layer. It is very easy to make a mistake if I removed the code and the system may become unstable. Moreover, the company also asked me to change the source code as little as possible in order to implement the new functionality. Then I gave up my plan.

At last, the implementation of this thesis was to keep the code in the application layer but to implement a new extra rdbms layer into the system. So the system would do double validation checks both in the application layer and in the rdbms layer. Later in order to evaluate the system performance, I could compare the running results on both Kred+rdbms system and Kred system. From the results, the evaluation was determined by seeing if rdbms slowed down the system running or not.

Kred system was a rather complicated and a huge system and in its DBMS, more than 100 tables were existed. Because my task in this thesis was focused on the evaluation of rdbms usage but not on the integration for the whole system, a portion of DBMS was enough to clarify the results. Then I determined to choose only several related tables involving into the evaluation instead of the whole DBMS. Four tables: person, invoice, estore and persons_data were the selections for this task. Table person stored the online custormers' information for the system and attribute pno (personal number) was the primary key. Table invoice documented invoice information of online purchases and attribute invno (invoice number) was the primary key. Table estore stored on-line shop information and attribute eid (estore id) was the primary key. Table pers_data represented a relationship between table person and table invoice.

Then I started to consider from the technique point of view, how to integrate rdbms into Kred system. The first thing to do was to activate rdbms as Mnesia activity callback module in the proper place. After reading Kred system source code, I found Mnesia database was initialized in file kdb.erl. Then I activated rdbms in this module, more precisely, Mnesia starting function `mnesia:start()` was replaced by `mnesia:start([access_module, rdbms])`.

Secondly, I needed to find a solution to cut off the connection between the kdb layer and Mnesia; build a new connection between the kdb layer and rdbms. This task was also not hard. In file kdb.erl, instead of calling the old function `mnesia:transaction(F)`, I adjusted it with `mnesia:activity(transaction, F, [], rdbms)`. This function was to call Mnesia executing a list of requests functions `F`. As a result, kdb would forward the requests to rdbms instead of Mnesia. Usually, Mnesia would return a tuple `{atomic, Result}` back if the update was successful. But rdbms would return `{nested_atomic, Result}`. Hence a series of code adjustments were made in order to adapt this new case. We needn't worry about building a connection between rdbms and Mnesia. Because in rdbms, after the verifications were carried out, it would automatically send the passed requests to Mnesia. Otherwise, the failure requests were blocked by rdbms and it would return an error message back to the previous layer. If the update was failed in Mnesia, rdbms could also detect the error and returned it back to its previous layer.

In the end, the data type and RI constraints for the four tables needed to be defined into rdbms. A separate file was written where I used the rdbms API functions to define the constraints properly. This file was called kred_rdbms.erl. These constraints must be inserted into rdbms at the exact time after the four tables were created.

- A problem to use rdbms

I found a serious problem when using rdbms. All the data in the DBMS would be lost completely if Mnesia was restarted again. I did not discover the problem until I started to integrate rdbms into Kred system. This was due to the reason that when I implemented the prototype, I never stopped Mnesia and restarted it again. However in the integration process, I found Kred system restarted Mnesia for several times in order to clean the old data and re-defined the schema information. At the end, I failed to fix this problem but found another solution to continue my study.

A Mneisa API function `Mnesia:dump_to_textfile(Backup)` was able to backup all that data into a text file `Backup`. Another function `mesia:load_textfile(Backup)` was able to load all the data from a text file `Backup`. This function was used into the Kred system. Before Mnesia was stopped, I added function mnesia:dump_to_textfile(Backup); After Mnesia was started again, I added function mnesia:load_textfile(Backup) in order to recover the data from the previous state. However it was a poor solution that only for this thesis evaluation. In order to use rdbms into a real system, a better solution needed to be found out.

Chapter 7

Testing and Evaluation

This chapter introduces one testing and two evaluations carried out on the Kred system. The first section is about the correctness testing on the Kred system in order to test if rdbms could detect constraints violations for the Kred system or not. In the second part, an evaluation is carried out in order to see how much rdbms would slow down the Kred system running. Later a supplementary evaluation is added to do a more deep evaluation.

7.1 Correctness testing

Correctness testing was separated into two tasks. The first one was to test if rdbms could detect a data type violation or not; the second one was to test if rdbms could detect an RI constraint violation or not. Because the application layer could check a data constraint violation, so the invalid requests would always be blocked by the application layer and never be forwarded to the rdbms layer. Hence the testing could not be carried out by enter with invalid requests. On the other hand, it was possible to feed Kred system with valid requests. But in rdbms I could define a fake data constraint that never be satisfied by the valid requests. Then the valid requests could go through the application layer but in rdbms they were expected to be blocked.

The applications and tools for this testing included the Kred GUI web page and a command terminal. The command terminal was used to define database constraints into rdbms; form the Kred GUI web page, I could registered a new account into Kred system. Figure 7.1 shows the testing workflow. In this figure, we considered that kdb layer belonged to application layer.

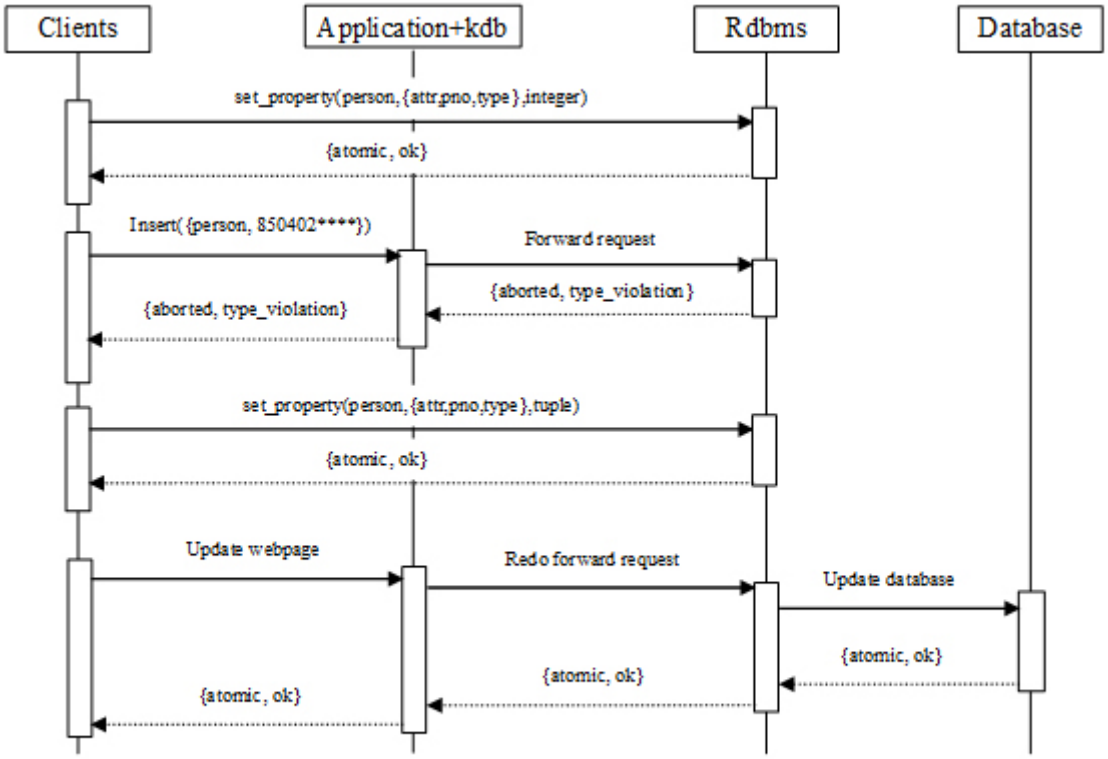


Figure 7.1 Rdbms detecting a data type violation

In the beginning, I deliberately defined a fake data type constraint into rdbms by calling rdbms_props:set_property(person, {attr, pno, type}, integer) from the command terminal. Actually, the value of data type of attribute pno should be tuple in the Kred system. Then from the Kred GUI web page, I registered a new account and clicked save to send this request which was shown in Figure 7.2.

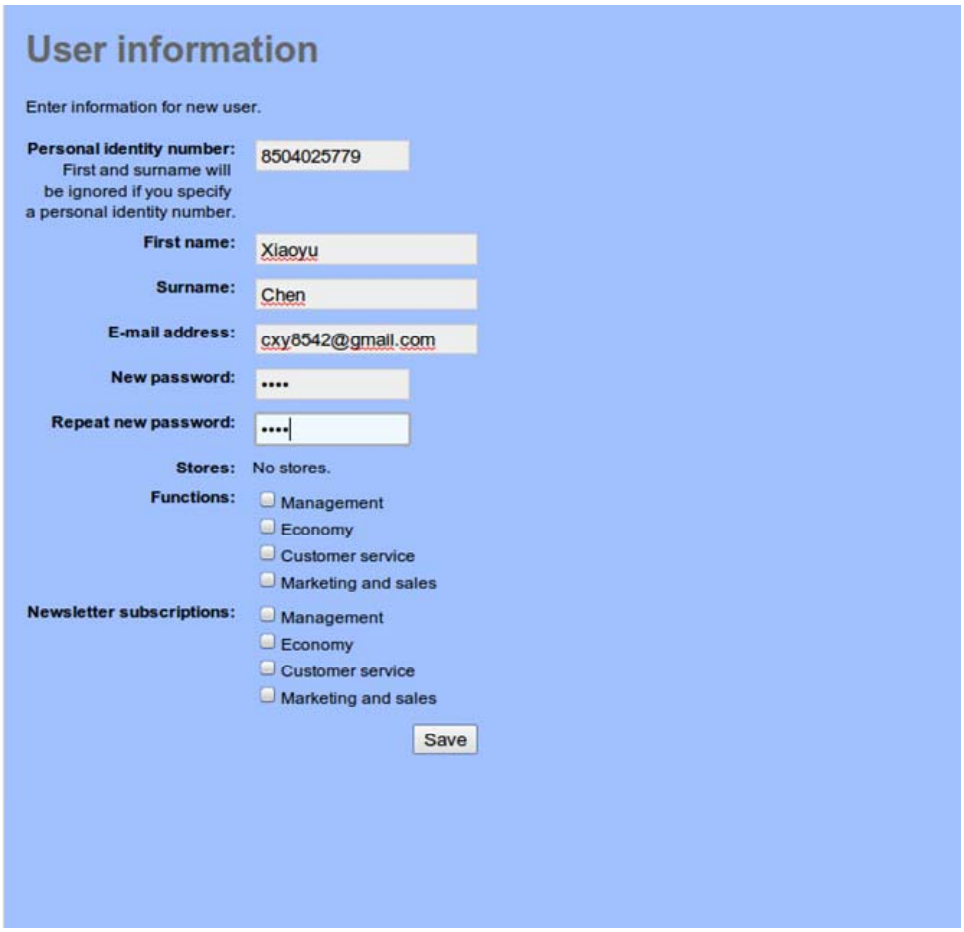


Figure 7.2 Register a new account

After submitting the registration, an error was returned and the request was aborted, as shown in Figure 7.3.

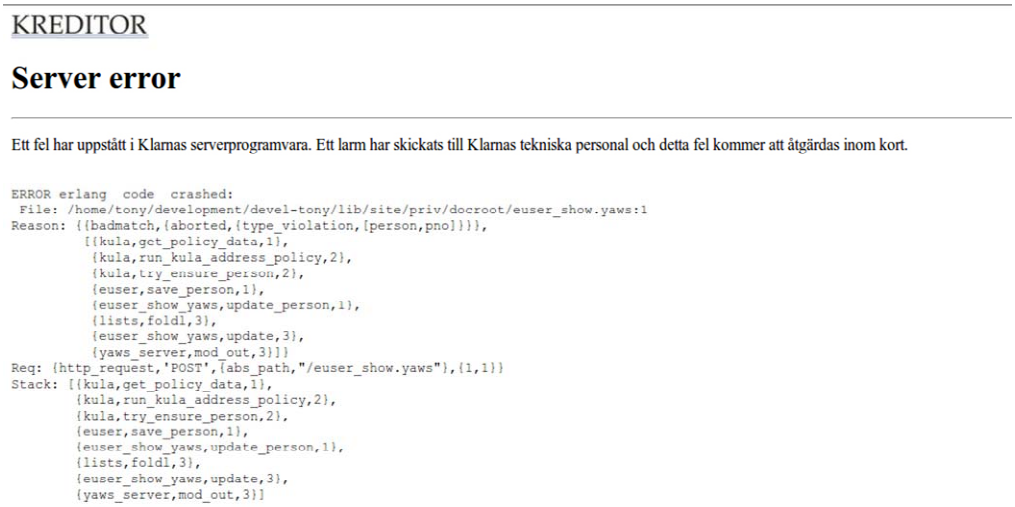


Figure 7.3 Data type violation error message

Then I defined the value of data type of attribute pno back to tuple into rdbms and refreshed the Kred GUI web page. This time, there was no error reported and the database was updated successfully.

In the second testing, the task was to test if rdbms could detect a RI constraint violation or not. The scenario was similar to the former one. Before registering a new customer into Kred system, I defined a fake relationship between the table person and the table invoice by calling `rdbms_props:set_property(person, add_references, [{attr, fname, [{invoice, invno, {full, ignore, no_action}}]}])` into rdbms. Then if there was an update into the table person, rdbms was expected to make sure that at least one object was existed in the table invoice where `person.fname` equaled to `invoice.invno`; otherwise the request was expected to be aborted. From the Kred GUI web page, I registered another new account. Then the system returned an error, shown in Figure 7.4.

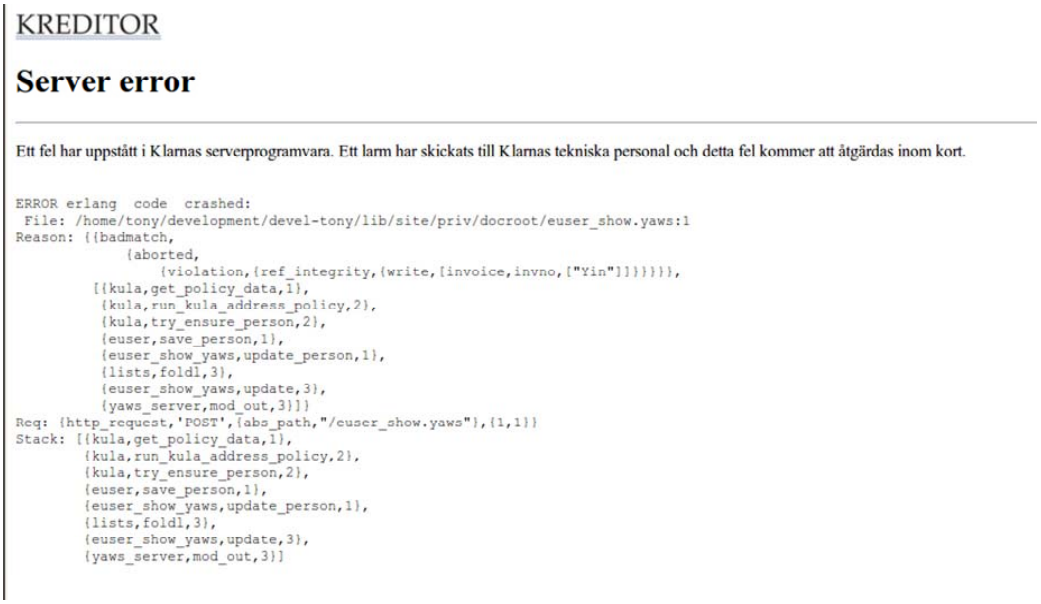


Figure 7.4 Referential integrity rule violation error message

Then I dropped the relationship by calling function `rdbms_props:set_property(person,drop_references,[{attr,fname,[{invoice,invno,{full,ignore,no_action}}]}])` and refreshed the Kred GUI web page. This time the request was successfully accepted.

The correctness testing proved that rdbms was truly effective in Kred system.

7.2 System performance evaluation

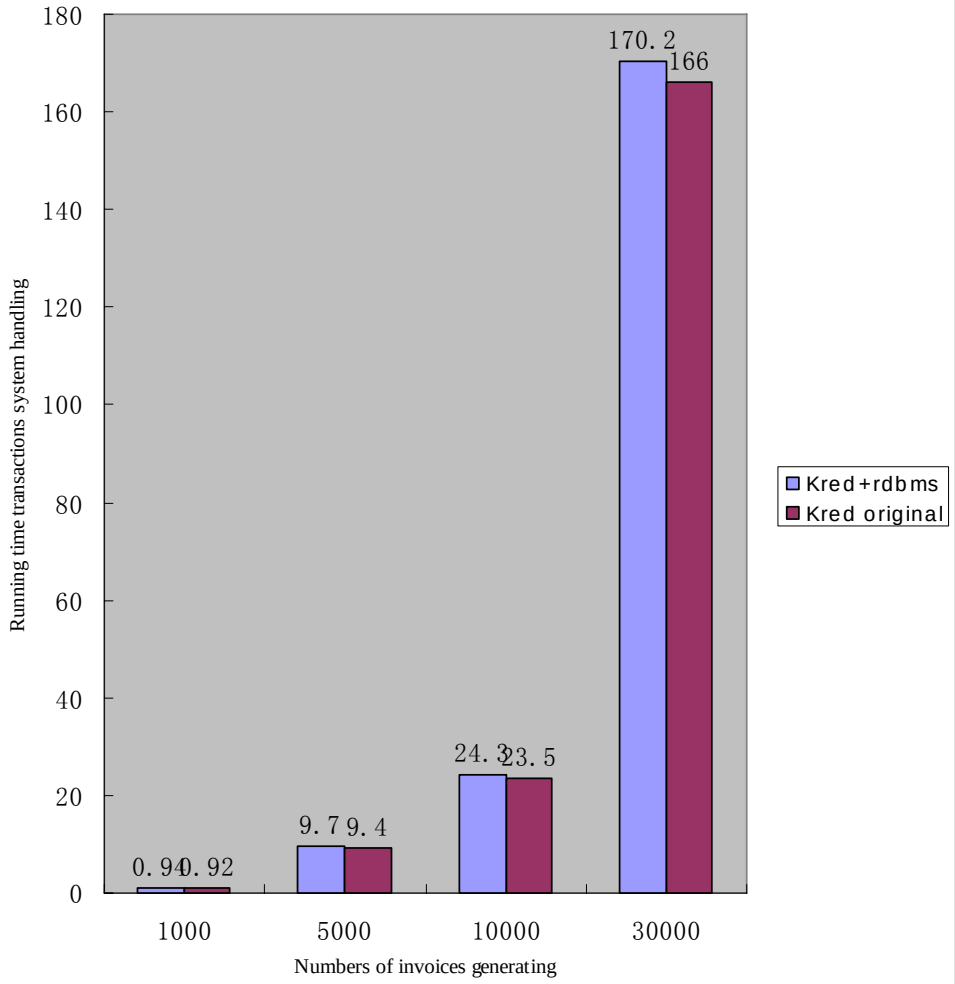
The idea for this evaluation was to execute the same number of table update transactions on both Kred+rdbms system and Kred system, calculate the executing time respectively and see if these two values varied much or not. If the difference was little, that might lead to a conclusion that rdbms was efficient, otherwise we could conclude that the rdbms was not efficient.

A Kred internal testing library file kerd_test_lib.erl was reused for this evaluation. This file provided a group of API functions to test Kred system. These testing functions were enable users to carry out table manipulation transactions into the DBMS. In this library, there was a function which helped to insert invoice object into the DBMS. Hence I decided to reuse this function to evaluate the system performance. Three parameters were necessarily required for this function: invoice number, the purchaser's personal number and the web store's name. I

implemented a function which could randomly generate a 9 digits invoice numbers. Then another function was written to randomly generate fixed numbers of personal numbers and the web stores' names. Then I finished the main testing file which could make invoices objects and insert them into the DBMS.

In order to obtain a long term result I choose 1000, 5000, 10000 and finally 30000 to be the numbers of generating invoices. For each level, the invoices were inserted into the Kred+rdbms and the Kred system. The following column chart shows the time running on both systems.

Column chart 7.1 Testing ruuning time of Kred+rdbms and Kred (minutes)



From the column chart, it was hard to say if rdbms really slowed down the system running or not, because the executing times running on Kred+rdbms and Kred system for executing the same number of transactions were almost the same. In the case of inserting 1000 invoices, Kred+rdbms system finished in 55.4 seconds as against 56.2 seconds for Kred system; in the long term case of inserting 30000 invoices into the system, the running time reached 170.2 minutes as against 166 minutes. Although the gap reached up to 4 minutes, compared with the whole running time, it was insignificant.

The column chart may suggest that the testing result was ideal. Rdbms running was too fast to be seen. However, it was possible that the testing result was affected by some undetected factors. I suspected that when the requests passed the application layer, it took too much time on executing the business logic and it may hide the time costing on rdbms operations. The evidence can be also seen from this chart. The time gap in the case of 1000 invoices was too small to be seen. It was only one 1 second. But when the invoice number was increased to 30000, the time gap became 4 minutes as the increasing rate rose sharply. It might because that as the size of database grew, the time spent on operations on the database system also increased dramatically. As a result, the time gap between these two systems became obvious.

To make this clear, it was necessary to carry out another test that concentrated on calculating the time lapse on database operations in order to evaluate rdbms in an accurate way.

7.3 Mnesia Benchmark evaluation

7.3.1 Introduction

Mnesia Benchmark is internal integrated measurement software. It offers a scientific measurement to calculate the number of transactions executed per second (TPS) by Mensia database system. Mnesia Benchmark can generate a proper network by connecting a set of Erlang nodes. Among these Erlang nodes, a single or several nodes are initialized as distributed database servers; a group of clients' nodes are created for generating requests. For each client's node, Benchmark can also create required number of processes as the requests generators. When the evaluation starts, The Benchmark will initialize both database server nodes and the requests generating nodes, establish the evaluation network. Then in a period of time, the Benchmark will generate number of requests from the clients and measure the operating results from the database

servers. Finally, it will terminate the whole system and print out the evaluation results. Figure 7.5 shows an example of a Mnesia Benchmark evaluation report.

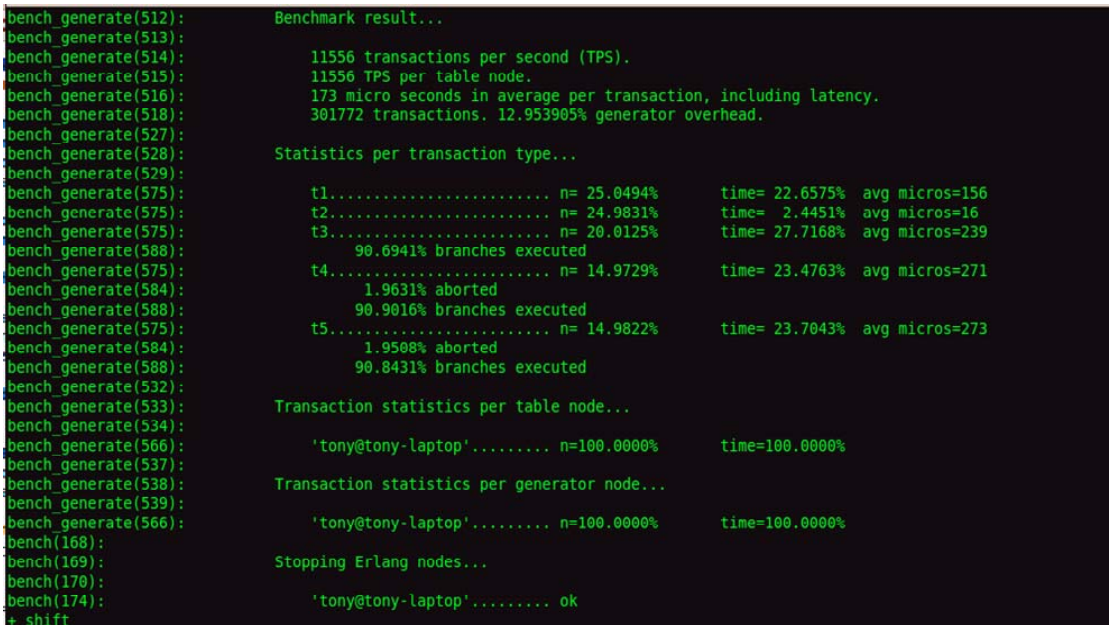

bench_generate(512): Benchmark result...
bench_generate(513):
bench_generate(514): 11556 transactions per second (TPS).
bench_generate(515): 11556 TPS per table node.
bench_generate(516): 173 micro seconds in average per transaction, including latency.
bench_generate(518): 301772 transactions. 12.953905% generator overhead.
bench_generate(527):
bench_generate(528): Statistics per transaction type...
bench_generate(529):
bench_generate(575): t1..... n= 25.0494% time= 22.6575% avg micros=156
bench_generate(575): t2..... n= 24.9831% time= 2.4451% avg micros=16
bench_generate(575): t3..... n= 20.0125% time= 27.7168% avg micros=239
bench_generate(588): 90.6941% branches executed
bench_generate(575): t4..... n= 14.9729% time= 23.4763% avg micros=271
bench_generate(584): 1.9631% aborted
bench_generate(588): 90.9016% branches executed
bench_generate(575): t5..... n= 14.9822% time= 23.7043% avg micros=273
bench_generate(584): 1.9508% aborted
bench_generate(588): 90.8431% branches executed
bench_generate(532):
bench_generate(533): Transaction statistics per table node...
bench_generate(534):
bench_generate(566): 'tony@tony-laptop'..... n=100.0000% time=100.0000%
bench_generate(537):
bench_generate(538): Transaction statistics per generator node...
bench_generate(539):
bench_generate(566): 'tony@tony-laptop'..... n=100.0000% time=100.0000%
bench(168):
bench(169): Stopping Erlang nodes...
bench(170):
bench(174): 'tony@tony-laptop'..... ok
+ shift

Figure 7.5 Benchmark teseing result example

In this report, the TPS result is calculated both for the whole network and for a single database sever node. It also calculates the percentage of different type of transactions generated during the evaluation. There are 5 types of transactions: t1, t2, t3, t4 and t5. T1 could represent as a single database read and t2 could be a single database write. All the types are fixed internally by Benchmark. But the Benchmark users are able to select one type, several type or randomly from the 5 type for the evaluation. Moreover, the report also calculates the percentage for both accepted transactions and aborted transactions during the evaluation.

Mnesia Benchmark can also support to generate a customized evaluation by set up a series of factors in a configuration file. Figure 7.6 shows an example of this Benchmark configuration file.

```
{cookie,                          bench_cookie}.
{generator_profile,               random}.
{statistics_detail,               debug}.
{generator_warmup,                1000}.
{generator_duration,              3000}.
{generator_cooldown,              1000}.
{generator_nodes,
 ['bench1@tony-desktop',
  'test@tony-desktop'
 ]}.
{use_binary_subscriber_key,       false}.
{n_generators_per_node,           2}.
{write_lock_type,                 sticky_write}.
{table_nodes,
 ['bench1@tony-desktop',
  'slave@tony-desktop'
 ]}.
{storage_type,                    ram_copies}.
{n_replicas,                      1}.
{n_fragments,                     10}.
{n_subscribers,                   5000}.
{n_groups,                        100}.
{n_servers,                       20}.
```

Figure 7.6 An example of benchmark configuration file

The Benchmark configuration file contains a list of factors. Factor **cookie** is used to build up the evaluation network. It prevents an unauthorized node to get access to the evaluation network. Factor **generator_profile** is used to select the type of generating transactions during the evaluation. It could be t1, t2, t3, t4, t5 or random. Factor **generator_warmup** defines the system warm up time before the evaluation starts. Factor **generator_duration** defines the actual time to evaluate the system. Factor **generator_cooldown** defines the system cool down time after the evaluation. All time should be in milliseconds. Factor **generator_nodes** defines a list of client generating nodes needed to be created for the evaluation. Factor **table_nodes** defines a list of database server node needed to be created for the evaluation. Factor **n_generator_per_node** defines the number of generating processes needed to be created on each node for the evaluation. Factor **storage_type** defines the database storage type. It could be ram or disc. Fragments, subscribers, groups and servers are the creating tables for the evaluation. Benchmark users can choose to insert a number of records into these four tables before the evaluation.

7.3.2 Benchmark evaluation with rdbms

The idea for this evaluation was to run the Benchmark on both Mnesia system and Mnesia plus rdbms integrated system with the same Benchmark configuration file. Then compare the two results in order to make a conclusion.

The following code is a description of the Benchmark configuration file running for this evaluation:

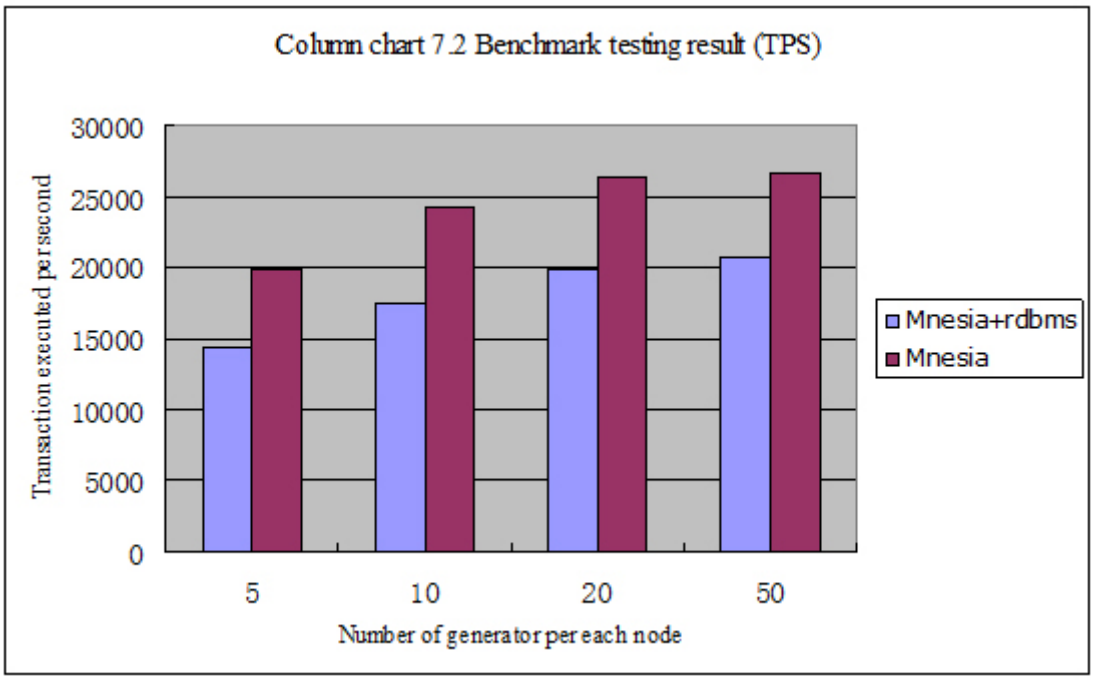
```
{cookie,                  bench_cookie}.
{generator_profile,           random}.
{statistics_detail,           debug}.
{generator_warmup,            50000}.
{generator_duration,          120000}.
{generator_cooldown,          50000}.
{generator_nodes,
 ['bench@tony-desktop'
 ]}.
{use_binary_subscriber_key,  false}.
{n_generators_per_node,      [5, 10, 20, 50]}.
{write_lock_type,      sticky_write}.
{table_nodes,
 ['bench@tony-desktop'
 ]}.
{storage_type,                ram_copies}.
{n_replicas,                  1}.
{n_fragments,                 1}.
{n_subscribers,                 500000}.
{n_groups,                      100}.
{n_servers,                     20}.
```

Random transaction type was selected for the evaluation because it was more close to the reality. 50 seconds (50,000 milliseconds) is selected for both system warm up time and cool down time. 120 seconds (120,000 milliseconds) was chosen for the evaluation time.

When I evaluated the Mnesia plus rdbms integrated system with Benchmark, I found it had some limitations. Firstly, the database was not able to be divided into fragments. Mnesia used an activity call back module mnesia_frag to handle this problem. Unfortunately, for each time, only one of these modules could be loaded by Mnesia and in this case, rdbms was necessary. Secondly, Benchmark could only generate one database server node and one generating node for this evaluation, otherwise the evaluation was failed. I tried to fix this problem, but I failed. This was really bad news for me because I planed to generate both light and heavy workloads to the database server by changing the number of generating

nodes in order to see if rdbms was impacted. However in this situation, the only way was to change the number of generator processes on each generating nodes.

The following chart shows the Benchmark result in which the creating generator processes vary from 5 to 50 per node.



From the chart, we can easily see that the capability of transaction handling of integrated rdbms Mnesia is only about three quarters as much as Mnesia. This time, the testing result is more obvious than in the former one. The chart shows explicitly that rdbms will slow down the system process. However it is still worth integrating rdbms into the system as the extra time taken is acceptable. The system also gains many other benefits from running rdbms.

From this chart, I also found an interesting phenomenon. TPS is increased dramatically when running 5 to 10 generator processes per node. However at 10 to 50, the increasing rate slows down very quickly. This indicates that sooner or later a top will be reached and after that TPS may stop increasing and either fluctuate or decrease.

Chapter 8

Conclusion

According to the research about rdbms during my thesis, I found rdbms was very useful software which could be easily integrated into Erlang/OTP implemented applications. It offered powerful functionalities to examine data type and RI constraints violations for Mnesia. The advantages of using rdbms include:

Fast integration procedure: Because rdbms was designed as a Mnesia activity call back module, it could be easily loaded and activated by using simple functions calling. Hence it saved precise time for the project development.

Stable performance: Rdbms was written by Erlang/OTP, so it inherited a set of effective error handling mechanisms from it. In addition, rdbms could also interact well with Mnesia. Hence rdbms integration will contribute to a robust system.

Efficiency: According to the evaluation form last chapter, it indicated that rdbms was efficient. In Kred system, the rdbms operation was almost not detected. Hence it was an efficient solution for Kred system.

On the other hand, the biggest problem in using rdbms is that rdbms could not retain old data when Mnesia is restarted no whether or not I keep the database in a ram memory or a disk. Since the data is precise for business purpose, it is urgent that this problem to be solved, otherwise rdbms are best avoided for use in a live system.

Future work:

A list of future work for extending my thesis is described below:

- Find the reason why rdbms could not retain data when restarted.
- Implement data constraints for the whole database within Kred system
- The rdbms manual indicates that, besides data constraints, rdbms has the ability to implement business logic. Hence a future work will be tried out to implement business logic inside Mnesia.

Acknowledgement

I would like to thank all those who helped me during the writing of my thesis. First of all, I am deeply thankful to my supervisor, Jonas Bergsten who was always patient to answering my questions. His encouragement and guidance played a key role in enabling me to finish my thesis. I felt shocked and very sad at his sudden passing away when the thesis was half finished. If it were only possible, I would like to express my heartfelt gratitude and share my happiness at finishing my thesis with him today.

Secondly, I would also like to thank Torbjörn Törnkvist, whose advice on how to write a good report was useful; Thanks are also due to Erik Stenman, who took the supervisor's responsibility after Jonas. He has helped me to finish the rest of the work. I would also like to thank my good friends and colleagues, Nicolae Paladi who gave me the permission to reuse his thesis result; Uwe Dauernheim whose creative ideas inspired me to implement several testing cases for my system; David Bingham who helped me to correct the English in my thesis.

Thirdly, I would like to thank Klarna AB who offered me this opportunity of doing my thesis; Thanks Uppsala University and especially my review Kjell Orsborn at the University, gave me helpful advice at the begining about how to plan the thesis work.

Finally, my gratitude goes to my parents who always support and encourage me especially during difficult time.

References

[1]. Website of business corporation, Klarna AB. http://www.klarna.se/sv/privat, last access date: February 21st, 2011.

[2]. Francesco Cesarini, Simon Thompson. Erlang Programming. O'REILLY, 2009.

[3]. Ericsson's website. http://www.ericsson.com/, last access date: February 21st, 2011.

[4]. Mnesia reference Manual. Ericsson AB, 2010.

[5]. Håkan. Mattson, Hans. Nilsson and Claes. Wikström. Mnesia-A Distributed Robust DBMS for Telecommunications Applications.

[6]. Thomas Connolly, Carolyn Begg. Database Systems, A Pratical Approach to Design, Implementation, and Management, Fifth Edition. Addison-Wesley, 2010.

[7]. Ramez Elmasri, Sham Navathe. Fundamentals of Database Systems, Fifth Edition. Addison-Wesley, 2006.

[8]. Wikipedia. Parent Table. http://en.wikipedia.org/wiki/Foreign_key, last access date: February 23rd, 2011.

[9]. Ulf Wiger. User's guide, Data Dictionary for Mnesia. Ericsson.

[10]. Wikipedia, Data Type. http://en.wikipedia.org/wiki/Data_type, last access date: February 21st, 2011.

[11]. Scott Ambler. Agile Database Techniqes: Effective Strategies for the Agile Software Developer (Wiley Application Development). Wiley, 2003.

[12]. Carlos Ordonez, Javier García-García, Zhibo Chen. Measuring referential integrity in distributed databases.

[13]. Wikipedia, Declarative Referential Integrity. http://en.wikipedia.org/wiki/Declarative_Referential_Integrity, last access date: February 22nd, 2011.

[14]. P. Srikanth. Oracle for Beginners Online Book. http://srikanthtechnologies.com/books/orabook/oraclebook.html, last access date: February 22nd, 2011.

[15]. Widom, J. and Cochrane, R. and Lindsay, B. Implementing Set-Oriented Production Rules as an Extension to Starburst. VLDB Conference, pages 275-285, 1991.

[16]. Xiaochun Yang, Guoren Wang. Mapping Referential Integrity Constraints from Relational Database to XML.

[17]. The Apache DB Project. http://db.apache.org/ojb/, last access date: February 22nd, 2011.

[18]. E. Simon, J. Kiernan, C. de Maindreville. Implementing High Level Active Rules on top of a Relational DBMS. 18th VLDB Conference, 1992.

[19]. Ulf Wiger's technique blog. http://ulf.wiger.net/weblog/, last access date: February 23rd, 2011.

[20]. Erlang, Mnesia Online Guide. Mnesia Activity description. http://www.erlang.org/doc/man/mnesia.html#activity-4, last access date: February 23rd, 2011.

[21]. Wikipedia. ACID. http://en.wikipedia.org/wiki/ACID, last access date: February 23rd, 2011.

[22]. Open-Source Erlang-White paper. http://www.erlang.org/white_paper.html, last access date: February 23rd, 2011.

[23]. Eric Kidd. XML-RPC HOWTO, 2011.